30｜Structured Streaming：从“流动的Word Count”开始

2021-11-19 吴磊

《零基础入门Spark》 课程介绍 >

讲述：吴磊
时长 15:49 大小 14.50M

你好，我是吴磊。

从今天这一讲开始，我们将进入流计算的学习模块。与以往任何时候都不同，今天的大数据处理，对于延迟性的要求越来越高，因此流处理的基本概念与工作原理，是每一个大数据从业者必备的“技能点”。

领资料

在这个模块中，按照惯例，我们还是从一个可以迅速上手的实例开始，带你初步认识 Spark 的流处理框架 Structured Streaming。然后，我们再从框架所提供的能力、特性出发，深入介绍 Structured Streaming 工作原理、最佳实践以及开发注意事项，等等。

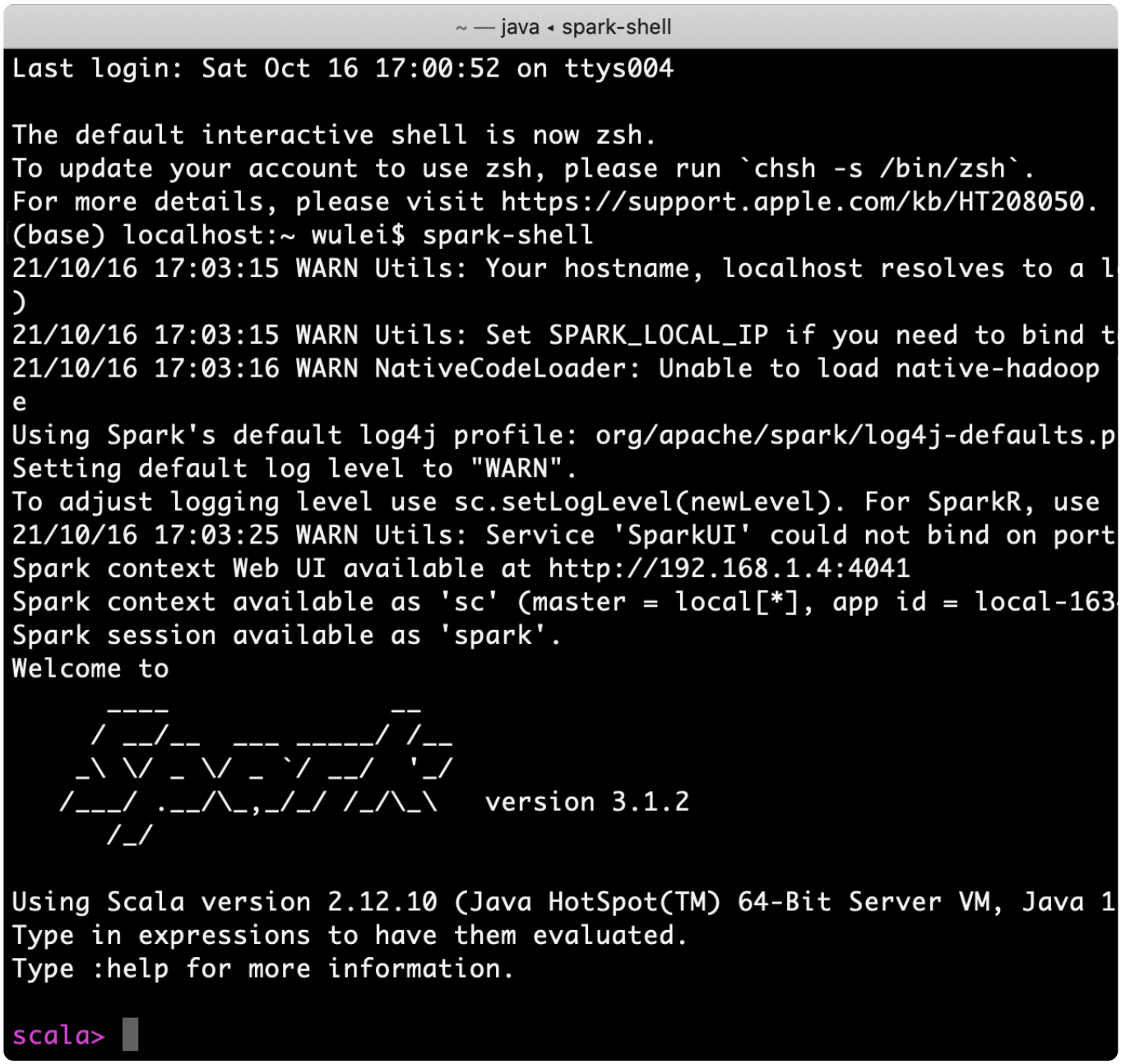
在专栏的第一个模块，我们一直围绕着 Word Count 在打转，也就是通过从文件读取内容，然后以批处理的形式，来学习各式各样的数据处理技巧。而今天这一讲我们换个花

样，从一个“流动的 Word Count”入手，去学习一下在流计算的框架下，Word Count 是怎么做的。

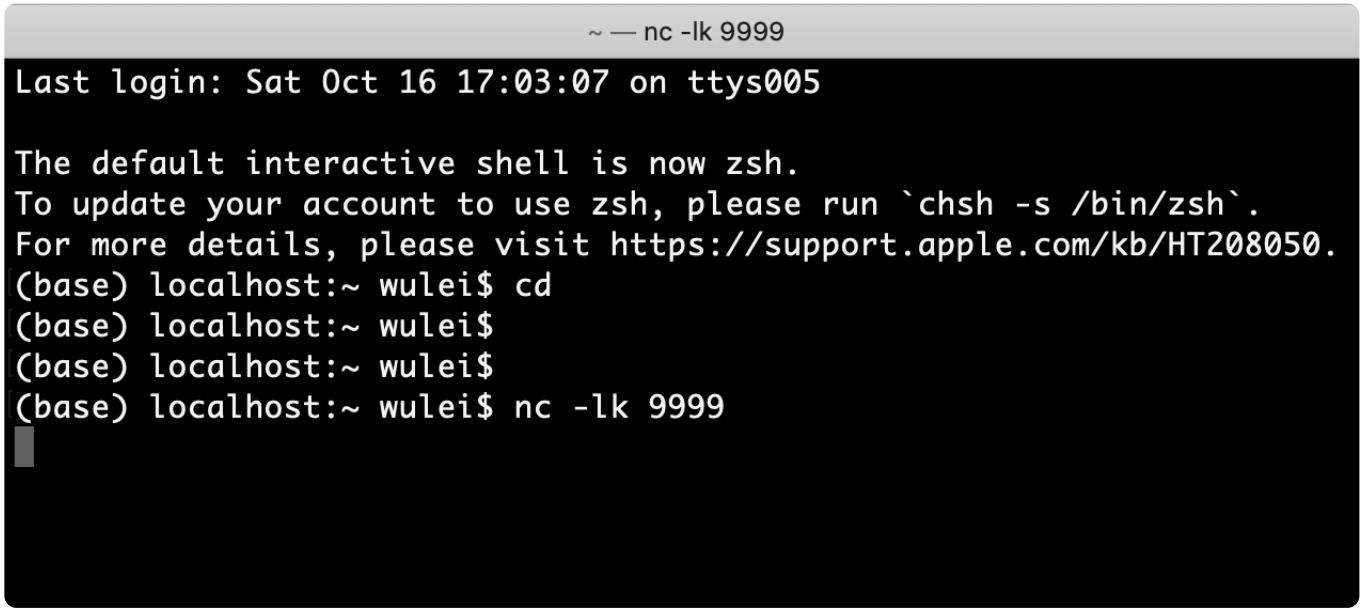
环境准备

要上手今天的实例，你只需要拥有 Spark 本地环境即可，并不需要分布式的物理集群。

不过，咱们需要以“流”的形式，为 Spark 提供输入数据，因此，要完成今天的实验，我们需要开启两个命令行终端。一个用于启动 spark-shell，另一个用于开启 Socket 端口并输入数据，如下图所示。



终端1：spark-shell & 终端2：netcat



终端1：spark-shell & 终端2：netcat

流动的 Word Count

环境准备好之后，接下来，我们具体来说一说，什么是“流动的 Word Count”。

所谓没有对比就没有鉴别，为了说清楚“流动的 Word Count”，咱们不妨拿批处理版本的 Word Count 作对比。在之前的 Word Count 中，数据以文件（wikiOfSpark.txt）的形式，一次性地“喂给”Spark，从而触发一次 Job 计算。而在“流动的 Word Count”里，数据以行为粒度，分批地“喂给”Spark，每一行数据，都会触发一次 Job 计算。

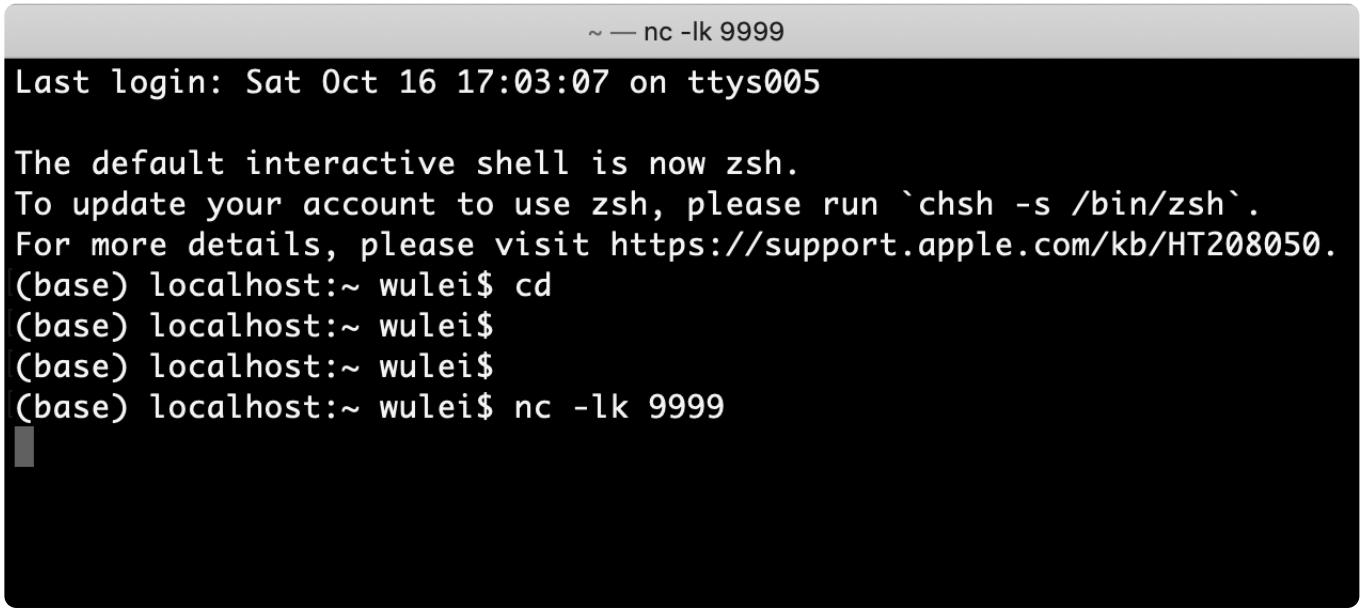
具体来说，我们使用 netcat 工具，向本地 9999 端口的 Socket 地址发送数据行。而 Spark 流处理应用，则时刻监听着本机的 9999 端口，一旦接收到数据条目，就会立即触发计算逻辑的执行。当然，在我们的示例中，这里的计算逻辑，就是 Word Count。计算执行完毕之后，流处理应用再把结果打印到终端（Console）上。

与批处理不同，只要我们不人为地中断流处理应用，理论上它可以一直运行到永远。以“流动的 Word Count”为例，只要我们不强制中断它，它就可以一直监听 9999 端口，接收来自那里的数据，并以实时的方式处理它。

好啦，弄清楚我们要做的事情之后，接下来，我们一起来一步一步地实现它。

首先第一步，我们在第二个用来输入数据的终端敲入命令“nc -lk 9999”，也就是使用 netcat 工具，开启本机 9999 端口的 Socket 地址。一般来说，大多数操作系统都预装了

netcat 工具，因此，不论你使用什么操作系统，应该都可以成功执行上述命令。


~ — nc -lk 9999
Last login: Sat Oct 16 17:03:07 on ttys005
The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
(base) localhost:~ wulei$ cd
(base) localhost:~ wulei$
(base) localhost:~ wulei$
(base) localhost:~ wulei$ nc -lk 9999

命令敲击完毕之后，光标会在屏幕上一直闪烁，这表明操作系统在等待我们向 Socket 地址发送数据。我们暂且把它搁置在这里，等一会流处理应用实现完成之后，再来处理它。

接下来第二步，我们从第一个终端进入 spark-shell 本地环境，然后开始开发流处理应用。首先，我们先导入 DataFrame，并指定应用所需监听的主机与端口号。

复制代码

```
import org.apache.spark.sql.DataFrame

// 设置需要监听的本机地址与端口号
val host: String = "127.0.0.1"
val port: String = "9999"
```

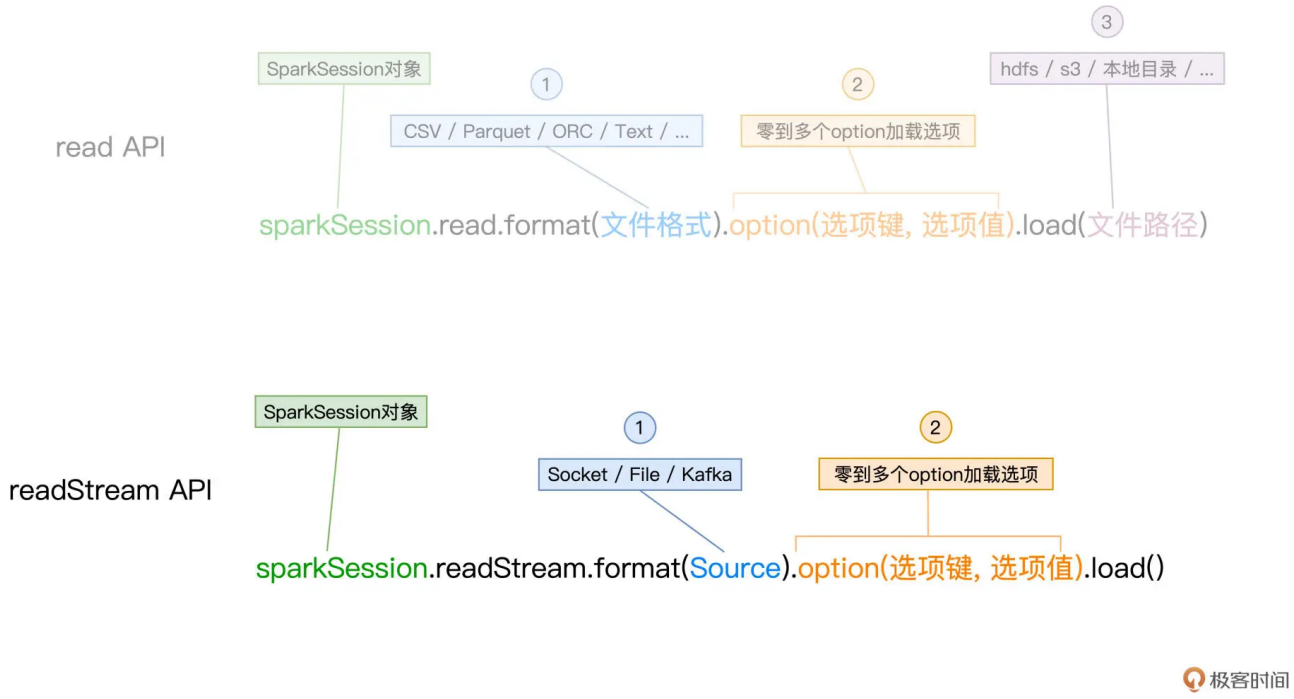
数据加载

然后是数据加载环节，我们通过 SparkSession 的 readStream API 来创建 DataFrame。

复制代码

```
// 从监听地址创建DataFrame
var df: DataFrame = spark.readStream
.format("socket")
.option("host", host)
.option("port", port)
.load()
```

仔细观察上面的代码，你有没有觉得特别眼熟呢？没错，readStream API，与 SparkSession 的 read API 看上去几乎一模一样。



readStream API与read API对比

可以看到，与 read API 类似，readStream API 也由 3 类最基本的要素构成，也就是：

format：指定流处理的数据源头类型

option：与数据源头有关的若干选项

load：将数据流加载进 Spark

流计算场景中，有 3 个重要的基础概念，需要我们重点掌握。它们依次是 Source、流处理引擎与 Sink。其中，Source 是流计算的数据源头，也就是源源不断地产生数据的地方。与之对应，Sink 指的是数据流向的目的地，也就是数据要去向的地方，后面我们讲到 writeSteam API 的时候，再去展开。

流计算基础概念

而流处理引擎是整个模块的学习重点，后续我们还会深入讨论。它的作用显而易见：在数据流动过程中实现数据处理，保证数据完整性与一致性。这里的数据处理，包括我们 Spark SQL 模块讲过的各种操作类型，比如过滤、投影、分组、聚合、排序，等等。

现在，让我们先把注意力放到 readStream API 与 Source 上来。通过 readStream API 的 format 函数，我们可以指定不同类型的数据源头。在 Structured Streaming 框架下，Spark 主要支持 3 类数据源，分别是 Socket、File 和 Kafka。

其中，Socket 类型主要用于开发试验或是测试应用的连通性，这也是本讲中我们采用 Socket 作为数据源的原因。File 指的是文件系统，Spark 可以通过监听文件夹，把流入文件夹的文件当作数据流来对待。而在**实际的工业级应用中，Kafka + Spark 的组合最为常见**，因此在本模块的最后，我们会单独开辟一篇，专门讲解 Kafka 与 Spark 集成的最佳实践。

通过 format 指定完数据源之后，还需要使用零到多个 option，来指定数据源的具体地址、访问权限等信息。以咱们代码中的 Socket 为例，我们需要明确主机地址与端口地址。

复制代码

```
// 从监听地址创建DataFrame
var df: DataFrame = spark.readStream
.format("socket")
.option("host", host)
.option("port", port)
.load()
```

一切准备就绪之后，我们就可以通过 load，来创建 DataFrame，从而把数据流源源不断地加载进 Spark 系统。

数据处理

有了 DataFrame 在手，我们就可以使用之前学习过的各类 DataFrame 算子，去实现 Word Count 的计算逻辑。这一步比较简单，你不妨先自己动手试试，然后再接着往下看。

复制代码

```
/**
使用DataFrame API完成Word Count计算
*/

// 首先把接收到的字符串，以空格为分隔符做拆分，得到单词数组words
df = df.withColumn("words", split($"value", " "))

// 把数组words展平为单词word
.withColumn("word", explode($"words"))

// 以单词word为Key做分组
.groupBy("word")

// 分组计数
.count()
```

首先，需要说明的是，我们从 Socket 创建的 DataFrame，默认只有一个“value”列，它以行为粒度，存储着从 Socket 接收到数据流。比方说，我们在第二个终端（也就是 netcat 界面），敲入两行数据，分别是“Apache Spark”和“Spark Logo”。那么在“value”列中，就会有两行数据与之对应，同样是“Apache Spark”和“Spark Logo”。

对于“value”列，我们先是用空格把它拆分为数组 words，然后再用 explode 把 words 展平为单词 word，接下来就是对单词 word 做分组计数。这部分处理逻辑比较简单，你很容易就可以上手，鼓励你尝试其他不同的算子，来实现同样的逻辑。

数据输出

数据处理完毕之后，与 readStream API 相对应，我们可以调用 writeStream API 来把处理结果写入到 Sink 中。在 Structured Streaming 框架下，Spark 支持多种 Sink 类型，其中有 Console、File、Kafka 和 Foreach(Batch)。对于这几种 Sink 的差异与特点，我们留到下一讲再去展开。

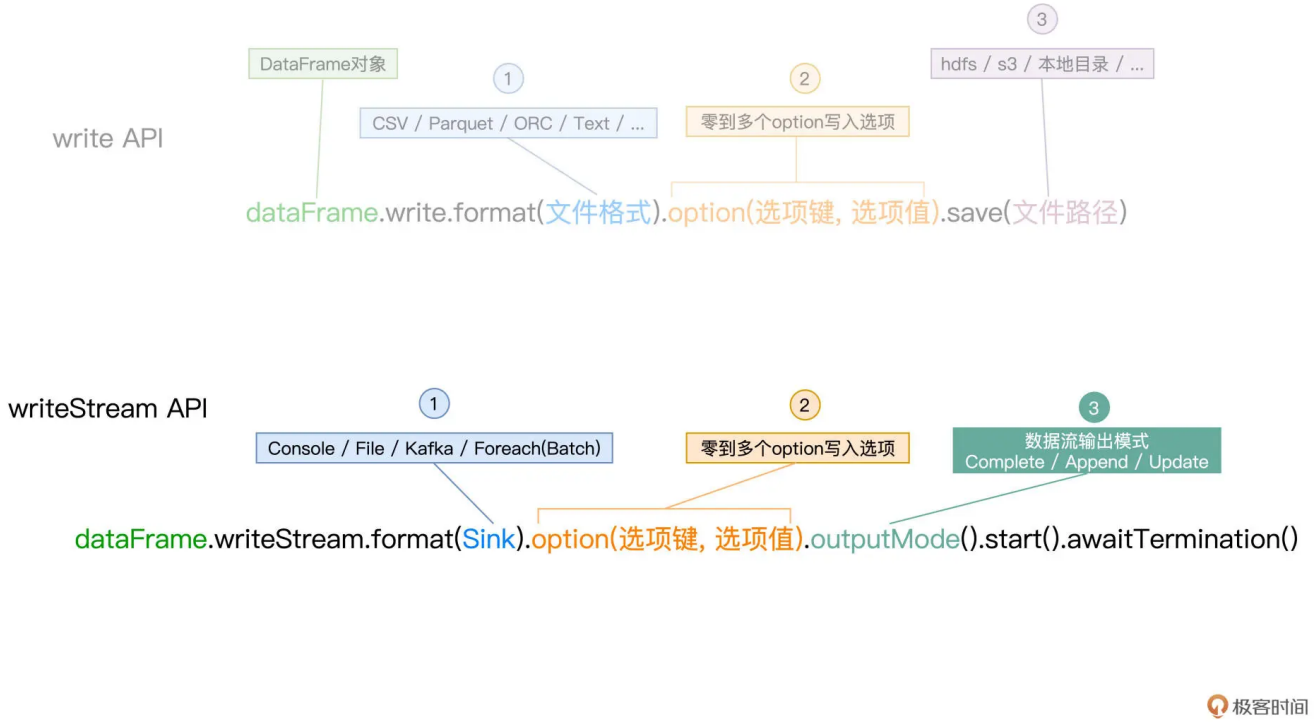
流计算基础概念

这里我们先来说说 Console，Console 就是我们常说的终端，选择 Console 作为 Sink，Spark 会把结果打印到终端。因此，Console 往往与 Socket 配合，用于开发实验与测试连通性，代码实现如下所示。

复制代码

```
/**
将Word Count结果写入到终端（Console）
*/

df.writeStream
// 指定Sink为终端（Console）
.format("console")

// 指定输出选项
.option("truncate", false)

// 指定输出模式
.outputMode("complete")
//.outputMode("update")

// 启动流处理应用
.start()
// 等待中断指令
.awaitTermination()
```

可以看到，writeStream API 看上去与 DataFrame 的 write API 也是极为神似。



writeStream API与write API对比

其中，format 用于指定 Sink 类型，option 则用于指定与 Sink 类型相关的输出选项，比如与 Console 相对应的“truncate”选项，用来表明输出内容是否需要截断。在 write API 中，我们最终通过调用 save 把数据保持到指定路径，而在 writeStream API 里，我们通过 start 来启动端到端的流计算。

所谓**端到端的流计算**，它指的就是我们在“流动的 Word Count”应用中实现的 3 个计算环节，也即从数据源不断地加载数据流，以 Word Count 的计算逻辑处理数据，并最终把计算结果打印到 Console。

整个计算过程持续不断，即便 netcat 端没有任何输入，“流动的 Word Count”应用也会一直运行，直到我们强制应用退出为止。而这，正是函数 awaitTermination 的作用，顾名思义，它的目的就是在“等待用户中断”。

对于 writeStream API 与 write API 的不同，除了刚刚说的 start 和 awaitTermination 以外，细心的你想必早已发现，writeStream API 多了一个 outputMode 函数，它用来指定数据流的输出模式。

想要理解这个函数，就要清楚数据流的输出模式都有哪些。我们先来说一说 Structured Streaming 都支持哪些输出模式，然后再用“流动的 Word Count”的执行结果，来直观地进行对比说明。

一般来说，Structured Streaming 支持 3 种 Sink 输出模式，也就是：

Complete mode：输出到目前为止处理过的全部内容

Append mode：仅输出最近一次作业的计算结果

Update mode：仅输出内容有更新的计算结果

当然，这 3 种模式并不是在任何场景下都适用。比方说，在我们“流动的 Word Count”示例中，Append mode 就不适用。原因在于，对于有聚合逻辑的流处理来说，开发者必须要提供 Watermark，才能使用 Append mode。

后面第 32 讲我们还会继续学习 Watermark 和 Sink 的三种输出模式，这里你有个大致印象就好。

执行结果

到目前为止，“流动的 Word Count”应用代码已全部开发完毕，接下来，我们先让它跑起来，感受一下流计算的魅力。然后，我们再通过将 outputMode 中的“complete”替换为“update”，直观对比一下它们的特点和区别。

要运行“流动的 Word Count”，首先第一步，我们把刚刚实现的所有代码，依次敲入第一个终端的 spark-shell。全部录入之后，等待一会，你应该会看到如下的画面：

```
21/10/17 16:52:04 WARN TextSocketSourceProvider: The socket source should n
ot be used for production applications! It does not support recovery.
21/10/17 16:52:05 WARN StreamingQueryManager: Temporary checkpoint location
 created which is deleted normally when the query didn't fail: /private/var
/folders/kz/b5rmzfwn0x3gml2ck_4ydn600000gn/T/temporary-7e0a630f-10f8-4f25-9
42e-65bcc92c030a. If it's required to delete it under any circumstances, pl
ease set spark.sql.streaming.forceDeleteTempCheckpointLocation to true. Imp
ortant to know deleting temp checkpoint folder is best effort.
-------------------------------------------

Batch: 0
-------------------------------------------
+----+-----+
|word|count|
+----+-----+
+----+-----+
```

“流动的Word Count”启动界面

当出现“Batch: 0”字样后，这表明我们的流处理应用已经成功运行，并在 9999 端口等待数据流的录入。接下来，我们切换到第二个终端，也就是开启 netcat 的终端界面，然后，**依次逐行**（注意！依次逐行！）输入下面的文本内容，每行数据录入之间，请间隔 3~5 秒。

行号	内容
1	Apache Spark
2	Spark Logo
3	Structured Streaming
4	Spark Streaming

极客时间

netcat输入的文本内容

然后，我们再把屏幕切换到 spark-shell 终端，你会看到 Spark 跑了 4 批作业，执行结果分别如下。

```
-------------------
Batch: 1
-------------------
+------+-----+
|word  |count|
+------+-----+
|Apache|1    |
|Spark |1    |
+------+-----+
```

```
-------------------
Batch: 2
-------------------
+------+-----+
|word  |count|
+------+-----+
|Logo  |1    |
|Apache|1    |
|Spark |2    |
+------+-----+
```

```
-------------------
Batch: 3
-------------------
+----------+-----+
|word      |count|
+----------+-----+
|Logo      |1    |
|Apache    |1    |
|Structured|1    |
|Streaming |1    |
|Spark     |2    |
+----------+-----+
```

```
-------------------
Batch: 4
-------------------
+----------+-----+
|word      |count|
+----------+-----+
|Logo      |1    |
|Apache    |1    |
|Structured|1    |
|Streaming |2    |
|Spark     |3    |
+----------+-----+
```

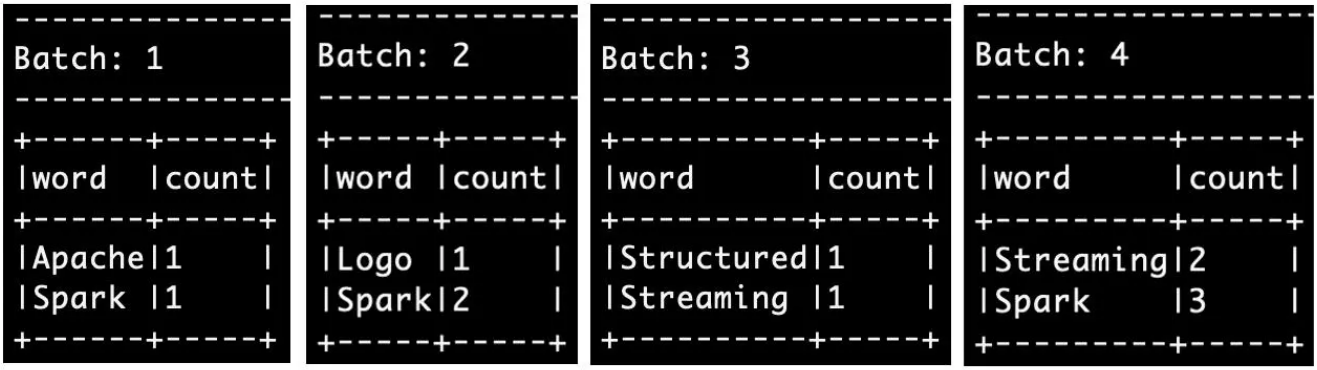
流处理应用的执行结果（Complete mode）

可以看到，在 Complete mode 下，每一批次的计算结果，都会包含系统到目前为止处理的全部数据内容。你可以通过对比每个批次与前面批次的差异，来验证这一点。

接下来，我们在 spark-shell 终端，输入强制中断命令（ctrl + D 或 ctrl + C），退出 spark-shell。然后再次在终端敲入“spark-shell”命令，再次进入 spark-shell 本地环

境，并再次录入“流动的 Word Count”代码。不过，这一次，在代码的最后，我们把 writeStream 中的 outputMode，由原来的“complete”改为“update”。

代码录入完毕之后，我们再切回到 netcat 终端，并重新录入刚刚的 4 条数据，然后观察第一个终端 spark-shell 界面的执行结果。



流处理应用的执行结果（Update mode）

对比之下一目了然，可以看到在 Update mode 下，每个批次仅输出内容有变化的数据记录。所谓有变化，也就是，要么单词是第一次在本批次录入，计数为 1，要么单词是重复录入，计数有所变化。你可以通过观察不同批次的输出，以及对比 Update 与 Complete 不同模式下的输出结果，来验证这一点。

好啦，到目前为止，我们一起开发了一个流处理小应用：“流动的 Word Count”，并一起查看了它在不同输出模式下的计算结果。恭喜你！学到这里，可以说，你的一只脚已经跨入了 Spark 流计算的大门。后面还有很多精彩的内容，有待我们一起去发掘，让我们一起加油！

重点回顾

今天这一讲，你需要掌握如下几点。首先，你需要熟悉流计算场景中 3 个重要的基本概念，也就是 Source、流处理引擎和 Sink，如下图所示。

极客时间

流计算基础概念

再者，对于 Source 与 Sink，你需要知道，在 Structured Streaming 框架下，Spark 都能提供哪些具体的支持。以 Source 为例，Spark 支持 Socket、File 和 Kafka，而对于 Sink，Spark 支持 Console、File、Kafka 和 Foreach(Batch)。

之后我们结合一个流处理小应用，借此熟悉了在 Structured Streaming 框架下，流处理应用开发的一般流程。一般来说，我们通过 readStream API 从不同类型的 Source 读取数据流、并创建 DataFrame，然后使用 DataFrame 算子处理数据，如数据的过滤、投影、分组、聚合等，最终通过 writeStream API 将处理结果，写入到不同形式的 Sink 中去。

最后，对于结果的输出，我们需要了解，在不同的场景下，Structured Streaming 支持不同的输出模式。输出模式主要有 3 种，分别是 Complete mode、Append mode 和 Update mode。其中，Complete mode 输出到目前为止处理过的所有数据，而 Update mode 仅输出在当前批次有所更新的数据内容。

每课一练

在运行“流动的 Word Count”的时候，我们强调**依次逐行**输入数据内容，请你把示例给出的 4 行数据，**一次性地输入 netcat（拷贝 & 粘贴）**，然后观察 Structured Streaming 给出的结果，与之前相比，有什么不同？

欢迎你在留言区跟我交流互动，也推荐你把今天的内容分享给更多同事、朋友，一起动手搭建这个 Word Count 流计算应用。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 0 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 29 | Spark MLlib Pipeline：高效开发机器学习应用

下一篇 用户故事 | 小王：保持空杯心态，不做井底之蛙

更多学习推荐

精选留言 (1)

写留言

大叮当
2021-11-19

老师您好，一直有个问题想请请您解惑下。
就是kafka每个主题都有个分区的概念，理论上说，一个消费者组中消费者数目和分区数一致，是效率最高的。
引申到Spark Streaming/Spark Structured Streaming,我的理解：
1、executor数目，和要消费的topic中的分区数目一致，效能最高，不知道这个点我理解...
展开